Using performance forecasting to accelerate elasticity

Paulo Moura[1] *, Spyros Voulgaris[2], and Maarten van Steen[3]

[1] University of São Paulo, Brazil
pbmoura@ime.usp.br
[2] VU University Amsterdam, The Netherlands
spyros.voulgaris@vu.nl
[3] University of Twente, The Netherlands
m.r.vansteen@utwente.nl

Abstract. Cloud computing facilitates dynamic resource provisioning. The automation of resource management, known as elasticity, has been subject to much research. Monitoring of a running service plays a crucial role, and adjustments are made when certain thresholds are crossed. On such occasions, it is common practice to simply add or remove resources. In this paper we ask ourselves how we can predict the performance of a service in order to dynamically adjust allocated resources based on predictions. In other words, instead of "repairing" because a threshold has been crossed, we attempt to stay ahead and allocate a best amount of resources in advance. To do so, we need to have accurate predictive models that are based on workloads. We present our approach, based on the Universal Scalability Law, and discuss initial experiments.

Keywords: cloud computing, elasticity, performance prediction, scalability modeling

* Contact info: P.O. Box 217, 7500 AE Enschede

1 Introduction

In this paper we address the following question: given the initial behavior of a service running in the cloud, can we forecast its required peak performance in order to preallocate enough resources so that it can meet those demands? This question is important when incrementally adjusting the allocation of resources to a cloud service does not suffice, or is simply too expensive.

In many cases, elasticity in the cloud is obtained by closely monitoring the current behavior of a service, and when certain thresholds are passed, adjustments are made. For example, a virtual machine is added or removed, the number of CPUs is changed, or the amount of memory is adjusted. However, monitoring a service and making adjustments comes at a price. For example, adding or removing a virtual machine may incur significant costs for transferring data between machines. For this reason, not only should we consider which thresholds to use for triggering an adjustment, but also the moments at which we are willing to make the costs for adjustments. Roughly speaking, when we accept changes after small time intervals, we can expect higher aggregated adjustment costs compared to the case in which changes are instantiated only after significant time has elapsed. The downside of the latter is obviously a waste of resources, or a degradation in quality of service when simply not enough resources have been allocated to sustain current demand.

Ideally, we would know exactly in advance what is going to be demanded from a service so that we can precalculate the required resources to meet those demands, but also take into account the costs of changing the allocation of resources. Under those circumstances, we could then devise a change scheme in which the trade-off between resource usage, costs of change, and attained performance can be balanced. As a step toward this ideal situation, we ignore fine-grained adjustments and focus on allocating enough resources in order to meet peak-performance demands.

Our approach requires an adequate predictive model by which we can compute the expected peak performance. In this paper, we discuss our experiences with one such model, the Universal Scalability Law (USL), developed by Neil Gunther [14]. In particular, we adopt his model and combine it with curve-fitting techniques taking only early performance samples from a running service. Fitting a curve to a USL model allows us to predict peak demands, and thus what is needed in terms of resources to ensure those demands can be met. As we report, USL has important limitations when applying it to cloud services. As it turns out, applications need to fit the USL framework rather strictly in order to use that framework for predicting resource usage. Nevertheless, when there is a fit, results are promising.

The paper is organized as follows. After briefly discussing related work, we move on to delving into some of the details that motivate our work, in Section 3. Our approach is discussed in detail in Section 4. We have run a number of experiments in the form of emulations and report our findings in Section 5, furthering discussion in Section 6, to conclude in Section 7.

2 Related Work

Support for elasticity is one of the key benefits offered by cloud computing. Cloud providers usually offer an API by which users can programmatically request resource allocation and deallocation on demand. Some also provide automated resource provisioning through an auto-scaling interface (e.g., Amazon Auto-Scaling[4]) where users can define rules, based on performance metrics, to automatically add or release resources. Alternatively, there are third-party tools for resource management automation, such as RightScale[5].

Along these lines, Chapman et al. [4] examine key requirements for service definition and propose a language to manage elasticity, defining a standard to support the federation and interoperability of computational clouds. This language can be used to describe service requirements and to provide rules on how to respond to performance and workload variation.

Other research focuses on identifying when and where to add or remove machines from a cloud system, applying feedback control. Aljohani et al. [1] propose a solution based on queuing theory. Its distinctive feature is that it considers that requests queue up in the application servers rather than in the load balancer. The model assumes a first-come-first-served policy and sets two thresholds to trigger the actions of scaling based on queue sizes.

Lim at al. [23] worked on proportional thresholds, that adapt based on cluster size to improve resource management. Dejun et al. [8, 7] propose a method in which only the front-end should receive a service-level objective. Every service is modeled as a queue, and resource provisioning or deprovisioning is performed after negotiation has taken place to identify *which* service it should be applied to.

Harbaoui and colleagues [17, 18, 26] propose to split the system up in a set of black-boxes, and to experiment with them to identify the appropriate queueing model predicting their performance. They, subsequently, compose a queueing network that identifies when a bottleneck appears and, in a decision process, chooses the best system configuration.

Elasticity acceleration was proposed based on historical evaluation. Gong et al. [10] use signal processing techniques to find patterns in workload and resource usage to speed up allocation of resources. When no pattern is identified by the signal processor, a discrete-time Markov chain takes place.

Vasić et al. [32] experimented with numerous off-the-shelf machine-learning techniques, reporting good results with Bayesian models and decision trees. These approaches rely on a feedback control loop to provide elasticity. In an initial phase, elasticity is based only on feedback control during data collection to build the model. Also, the model cannot predict resource demands for not-yet-observed load levels.

[4] `https://aws.amazon.com/autoscaling/`

[5] `http://www.rightscale.com/solutions/problems-we-solve/cloud-availability`

3 Motivation

Taking educated decisions on the amount of resources to allocate to running systems is essential to their uninterrupted high-performance operation. There are two main classes of models for achieving that. This section discusses these two classes, and motivates our proposed methodology on addressing cloud elasticity and scalability.

3.1 Elasticity

Elasticity is obtained by means of a control component that constantly monitors the running system. Measurements are compared with the values or ranges as specified in elasticity rules. Whenever a threshold is surpassed, an action is triggered to update the system's configuration. When performance is low or consumption is high, more resources are included into the system, and when consumption is low, resources are released. A common characteristic of most existing proposals is that every time an action is performed to update a configuration, a predefined number of resources (frequently only one) is added (or released).

Following the common approach, when there is a substantial increment in the workload the system may need to go through a sequence of measure-trigger-update cycles. The time needed to properly configure the system will be longer and, in the meantime, performance may degrade. Conversely, knowing beforehand how many resources a given workload requires, all resources can be allocated in a single action, speeding up the procedure and benefitting performance maintenance. In other words, it may be better to opt for future situations than repairing for the present.

However, to do so, a *predictive model* should be devised to infer the relation between workload and resource demand. Next we elaborate on how these models are conceived.

3.2 Scalability modeling and evaluation

There are two prominent classes of models for system performance and scalability. Analytical models [3], based on queuing theory and stochastic processes, are usually applied in early development stages based on architectural specifications. They can be used to obtain performance and scalability predictions that can guide architectural refinements. Queues have parameters to specify the distribution and frequency of arriving requests, distribution and mean execution time, system capacities in terms of waiting queue length and parallel processing. A model can be composed of a set of interconnected queues. The model itself can also be refined as system development advances, notably when more information is available to set queue parameters.

While analytical models require knowledge about system internals, curve-fitting models rely only on external observations of system behavior. Such observations are obtained by measuring metrics of interest. However, curve-fitting

models require a running system in order to be able to measure what is going on. A dataset of workload and performance metrics is analyzed by means of statistical inference to obtain a function that relates the selected metrics.

Models are traditionally used for capacity planning [24], but they are now also being applied at runtime to automate resource management, providing elasticity [8, 17, 31].

The precision of analytical models is limited by their inherent degree of abstraction, while precision of curve fitting is limited by variability of measurements. Precision and applicability of curve-fitting models may also be affected by underlying assumptions. For example, there are approaches that rely on segmenting a curve. This segmentation can be done by using adaptive splines [6, 16] or by splitting the model in two or more functions when different patterns are identified [2, 7]. This approach limits the model to the data space covered by the measurements. Other approaches make assumptions about system characteristics and how they affect performance [15, 28, 29]. The model has a particular shape and predictions beyond measured values are possible, as long as the model's assumptions continue to hold.

Our research relies on curve-fitting models. Thus, we are also concerned about obtaining data for modeling, notably in light of the fact that automation of gathering data points can be complex and time consuming. Curve-fitting comprises deploying the system, generating requests, collecting data about workload and performance, repeating those steps a number of times with different architectural configurations and request patterns, and subsequently analyzing performance output.

There are tools and frameworks to deal with this task [5, 21, 27, 30]. The caveat is that they are limited regarding automated analyses. Some simply store data, letting the analysis completely to the user [27]. Others offer a limited support by automating metric calculation [5] or plotting charts [21]. Yet in these cases, interpretation of the output is still left to the user.

We aim to automate fast scaling by applying a scalability model, as well as automate the modeling, as detailed in next section.

4 Proposal

As a first step we are working on a software framework to automate scalability evaluation of distributed systems [25]. The intent is to provide a tool to simplify definition and execution of scalability experiments. It includes software-extensible templates (such as abstract classes) to define how to communicate with the system under evaluation, how to change the workload at different steps, how to scale the system at different steps, and how to analyze the produced performance data. A set of implementations for these templates are being developed to simplify setting up experiments, but users are able to provide their own implementation that fit their needs as well.

We address the provisioning of meaningful and self-contained automated analysis. Common metrics proposed for scalability evaluation assume linear scal-

ing [5, 11, 9, 19, 22], yet many use arbitrary thresholds for qualifying a system to be scalable [20]. Our aim is to provide components to perform automated analysis in order to verify if system performance remains the same when resource allocation is changed because of variatons in workload [25]. Our basis is deriving a model that captures the relation between workload and resource demand. Deriving such a model is at the core of this paper.

Considering that a system is composed of a collection of communicating services, each service should have a performance level objective. For front-end services, this objective could be defined according to personal needs, agreement with clients, or other indicators. Internal services should have their performance goal established in a way that front-end service objectives can be met. All objectives must consider the capacities of a service. With objectives defined, one must identify the maximum workload that the system is able to support under a specific allocation of resources, and how to scale the system by changing the allocation of resources when the workload varies.

When doing experiments to evaluate and model the scalability of a given service, it is necessary that the services it communicates with reply according to their performance objectives. In some cases, remote mock-up objects, emulating such services [12], could be used to simplify experiment set-up.

We selected the Universal Scalability Law (USL) [13, 15] as starting point to provide a *predictive* scalability model. USL predicts a performance peak after which system performance is assumed to degrade. Since the USL model assumes no architectural restrictions, it should, in principle, be applicable to either multi-core, multi-processors, or distributed systems. The only constraint is that the architecture must be uniform, that is, homogeneous in its components.

USL assumes that performance can be improved via parallel processing, with the usual limitations. One such limitation is the assumption that certain parts of an execution are necessarily sequential. In particular, an execution is assumed to interchange between parallel task processing and sequential processing. The sequential portion is typically concerned with managing multiple processes, splitting data for parallel handling, or merging parallel execution outcomes. A sequential portion incurs **contention delays**. Fig. 1 shows how contention limits speedup obtained from parallelism. If there is no contention, changing a system architecture from one to four processes brings down execution time to one quarter. With contention, the reduction in execution time is less. Also, contention limits how much the system can be sped up through parallel processing because it does not improve the execution time of sequential portions.

Next to contention, we need to deal with data exchange between parallel executing tasks, referred to as **coherency**. Coherency delays are caused by the need to bring shared, yet replicated data into a consistent state. These delays happen at different levels, from CPU caches to remote storages. When processes need to write to a shared resource – be it a variable in local memory or a file on a remote disk – there is extra time needed to ensure data consistency. Coherency increases the execution time of each parallel process, as depicted in Fig. 2. The higher the degree of parallelism, more processes each process must synchronize

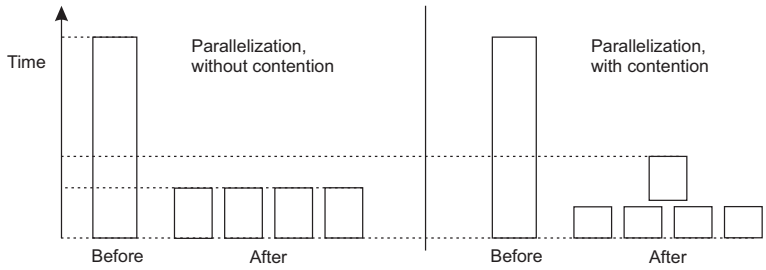

Fig. 1. The effect of contention delays on speedup.

with. Coherency grows quadratically with the number of parallel processes. At a certain point, this penalty will cause the total execution time to grow. From that point on, increasing parallelism degrades performance, instead of improving it.

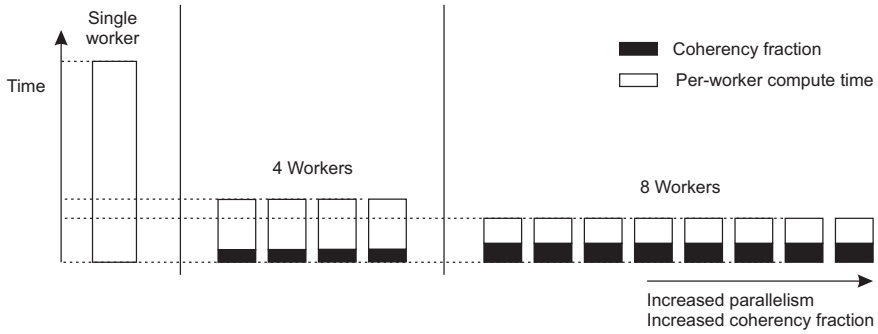

Fig. 2. The effect of coherency delays on speedup.

According to the USL model, the relation between performance and parallelism is ruled by the following formula:

$$C(p) = \frac{p}{1 + \sigma(p-1) + p\kappa(p-1)},$$

where p is the number of parallel processes, σ is the contention factor and κ is the coherency factor. C stands for the capacity and is obtained by normalizing the throughput reached with p processes, divided by the throughput of only a sequential execution. Contention and coherency are measured as the fraction of the sequential execution time. A value of 0 means that there is no effect on performance. A contention factor of 0.2, for instance, means that 20% of the sequential execution time cannot be parallelized. A coherency factor of 0.01 means that the time spent in the synchronization between two processes is 1% of the sequential execution time. The number of processes that provide maximum

throughput is as follows:

$$p_{max} = \sqrt{\frac{1-\sigma}{\kappa}}.$$

The USL model is claimed to be also valid when the architecture is fixed and the number of processes replaced by the number of concurrent users [15].

5 Experimental Evaluation

As we are mainly concerned at this point to validate the USL model for services running in the cloud, a set of relatively simple experiments were conducted. In particular, we are interested to see if USL can be used for *predictive* modeling that would allow us to allocate enough resources to sustain a peak workload.

We ran experiments on a large cluster of machines running CentOS 6, each having two quad-core Intel E5620 CPUs running at 2.4GHz, 24 GB of main memory, and interconnected via Gigabit Ethernet and InfiniBand interfaces.

5.1 Setup

For this first phase of evaluations, we are working with simple setups implemented in C to simulate workload execution. The execution is simulated by a busy-wait loop implemented as the `work` function below:

Busy wait loop

```
void work(int units, int usage, int delay) {
    int i, j;
    for (i = 0; i < units; i++) {
        for (j = 0; j < usage; j++)
            ;
        usleep(delay);
    }
}
```

The execution alternates between running an empty loop and sleeping. The argument `delay` sets the duration of each sleep in microseconds. The parameter `usage` sets how many iterations to run the empty loop, indirectly defining its duration. The relation between both duration slots defines the CPU utilization of the execution. The argument `units` is used to define the duration of the execution, setting how many times to alternate between the busy-wait loop and sleeping. A series of executions of `work` with different parameters were measured to identify desired values to use in the setup.

The setup consists of three kinds of nodes. One *Coordinator* receives requests to iterate over the busy-wait loop (i.e., the outer loop of `work`) a certain number of times. The workload is split among a set of *Workers*. Each *Worker* runs the busy-wait loop to simulate workload execution and communicates with the

Synchronizer. The latter also runs a busy-wait loop per received request to simulate a synchronization time among the *Workers* and replies. Note that this synchronization reflects the time, per worker, needed to bring shared data in a consistent state, thus capturing a coherency delay. The *Synchronizer* receives an initialization parameter specifying how many iterations to do as its busy wait. Thereafter, the *Workers* reply to the *Coordinator*. The *Coordinator* has a parameter related to the degree of contention, that is the fraction of the workload that is not split among the *Workers*, but executed by the *Coordinator* after receiving output from all the *Workers*. This execution flow is depicted in Fig. 3.

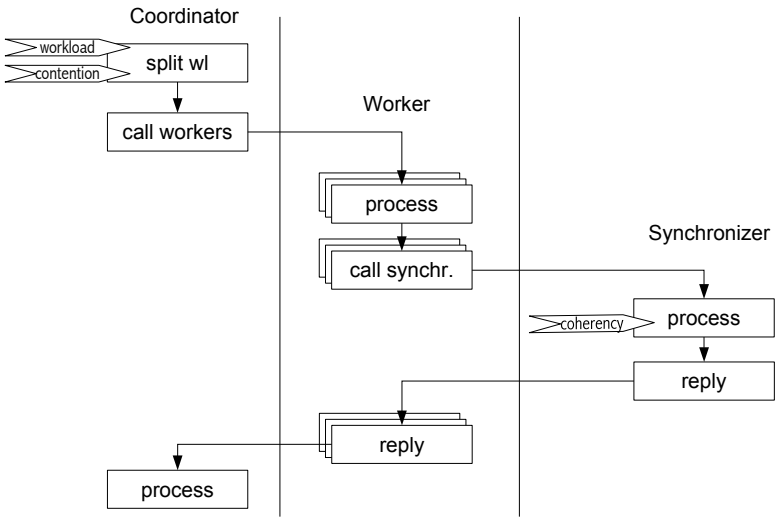

Fig. 3. Setup execution flow.

The experiments are executed in a sequence of requests with an increasing number of *Workers* to handle them. The other parameters are kept constant. The execution time of each request is measured and throughput calculated as *workload/time*, where *workload* is the number of times the loop was iterated. An R script was written to estimate the model.

5.2 Single Request

The first experiments with this setup were executed with a single request being sent per time. Thus, the *Coordinator* and *Workers* run a single process each, while the *Synchronizer* runs one process per *Worker*.

In most of the experiments, using the first six measurements were enough to obtain good models and the performance peak was between 10 and 28 *Workers*.

Fig. 4 is an example of the performance and model of an experiment with a workload of 10,000 iterations, with a contention fraction of 20% and coherency of 0.3%. The vertical line shows the last measurement used to fit the model. Peak performance occurs with 16 *Workers*.

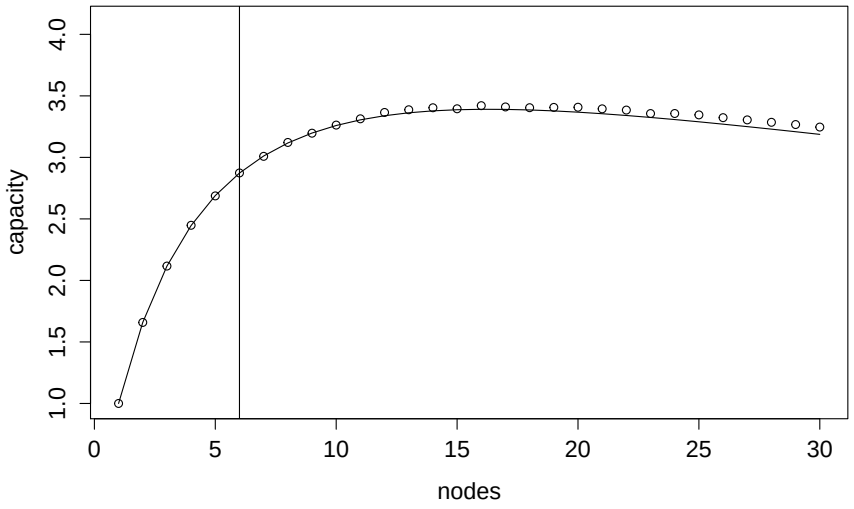

Fig. 4. Capacity variation with a single request and related model.

In an experiment with lower impact of contention (10%) and coherency (0.05%), with performance peak at 43 *Workers*, the first eight measurement were required for a better model. In all cases, the estimated model parameters were very close to the setup parameters.

5.3 Simultaneous Requests

Following the single request experiments, we executed a series of experiments with simultaneous requests being sent to the setup. In this case, the *Coordinator* and *Workers* run simultaneous parallel processes - one per request.

Performance degrades with the number of simultaneous requests even when there are enough resources to properly execute the workload. Figure 5 shows the performance curves of execution with one, three, and five simultaneous requests, with the same parameters of the experiment shown in Fig. 4. It is still possible to get good curve fitting, but more data must be used. As seen in Fig. 6, for an experiment with five simultaneous requests, a workload of 30,000 iteration per request, a contention fraction of 10% and a coherency of 0.1%, a good fitting was

achieved with 14 measurements. Discarding the performance with 23 *Workers*, when connection errors occurred, the peak performance was with 28 *Workers*.

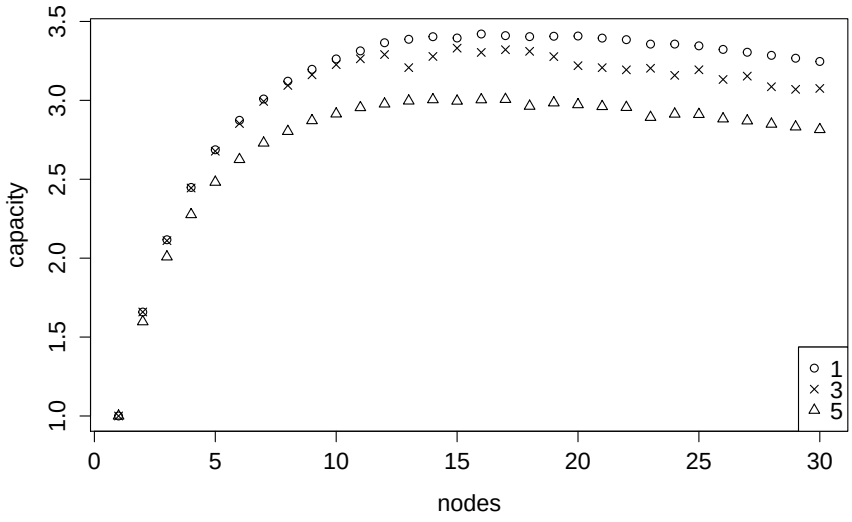

Fig. 5. Performance variation with one, three, and five simultaneous requests

We also observed that with the increase in the load on the *Synchronizer*, when it saturates, the degradation is faster than what the model predicts. It happened because time spent simulating each synchronization was affected by the time processes were put in wait for a processor. This is a limitation of this setup, which was implemented in this way for simplification and is not necessarily how synchronization would happen in practice. But it also reaffirms that the time spent with synchronization by each process must be linear to the number of processes, according to the model.

6 Discussion

Results observed so far show that the USL can be accurate under certain conditions and a deeper investigation on its applicability for cloud services seems worth the trouble. What is needed are more experiments exploring the different circumstances in which USL can, or should not be applied. The advantage of working with the current setup (Section 5.1) is the flexibility to change behavior in terms of request duration, CPU consumption, and the effects caused by parallelism and data sharing. The experiments executed with single requests

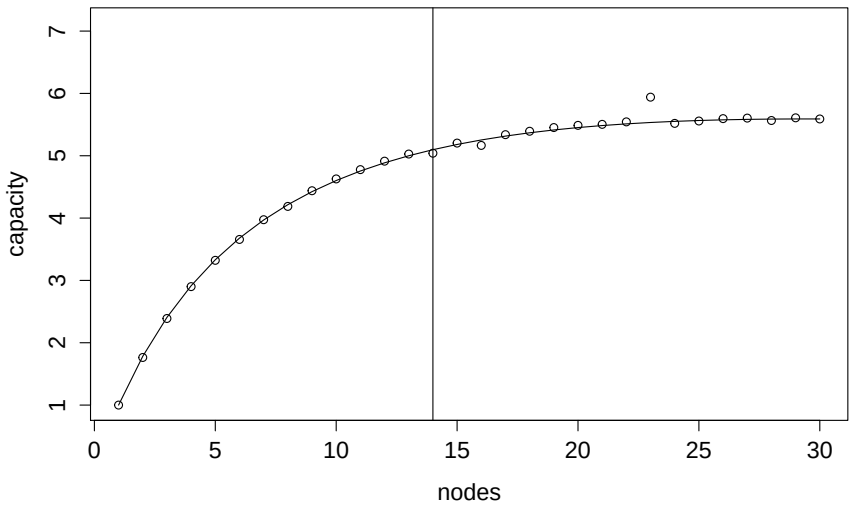

Fig. 6. Capacity variation with five simultaneous requests and the related model.

(Section 5.2) showing very accurate predictions and the estimated model parameters being in accordance with the experiment parameters show that the setup has the desired behavior. An important follow up will be to run experiments in similar conditions to those presented here, but adding variability to parameters. It would lead to observations, for instance, if having simultaneous processes with different durations or demanding different CPU load affect predictability. Furthermore, experiments with real systems are eventually imperative.

Regarding the experiments so far executed with multiple simultaneous requests, we observe that performance curve changes with the level of parallelism, as seen in Fig. 5. Our setup is comparable to a batch system with the number of iterations it runs being related to batch size. The model would be used to predict how many *Workers* should be employed in the execution to obtain best performance. But it would only provide correct estimates if the number of execution streams are the same as used in modeling. If the model was inferred with a single request, it will fail to predict demands when the system is processing three requests simultaneously, for instance. Thus, this is only practical if we limit the system to process one request per time. This is not always possible, hence we need to investigate the deduction of a model that is valid for an arbitrary level of parallelism.

The variation in node performance with load is due to internal contention. For instance, a *Worker* opens a socket and gets into a loop where it is waiting to accept an incoming connection. On receiving a connection it forks: the child process reads the workload from the socket, runs the busy wait and terminates;

the main process loops back to accept a new connection. Thus, there is a serialization in accepting connections. Since the *Coordinator* triggers the executions in parallel, a contention happens on a *Worker*'s accept.

The effects of such internal contention (and coherency) can be evaluated running experiments in a single machine. We executed experiments with two variations of the described setup. In the first one, *Coordinator* and *Synchronizer* were merged and the interprocess synchronization was implemented with shared semaphores. In this case, the performance curve escaped from the pattern imposed by the USL and observed in the experiments presented in Sec. 5. Afterwards, the semaphores were replaced by sockets, working as in the distributed executions. In this case, the results are comparable to those presented in the previous section. We believe that the difference is due to different dynamics related to shared memory access. But it is not clear how it would affect performance predictability of real systems.

We also tried to observe the relation between the arrival rate and the performance, executing experiments sending requests with a linearly increasing rate to a setup running in one node. In this case, the performance curve did not obey the USL. Roughly speaking, we observed an increasing throughput followed by a degradation. But the observed curve begins as a straight line while the service time is lower than the interrequest interval and starts bending when concurrency starts to occur. Also, the concurrency level grows faster at higher request rates, making the performance curve more severe. In these cases, we were able to obtain a reasonable fit using a subset of the measurements, but were unable to predict the curve by just sampling at the beginning of an experiment. Hence, we conclude that the arrival rate in not an adequate parameter to base the decisions related to resource allocation. As the previously presented experiments suggest, the decisions should be based on current system load. The control system must keep track of request arrivals and replies to account how many requests are being processed in the system at a given moment and use this information to set an appropriate resource allocation scheme.

7 Conclusions

Cloud computing has been gaining increasing adherence with one of its major appeals being the facility to auto-scale systems. Much research has been focusing on providing elasticity by reacting to variation in performance and utilization. In this research we examine another approach, where resource management is based on system load and a predictive model from which we can retrieve the resource demand of a given workload.

We presented preliminary evaluations of the applicability of the Universal Scalability Law to achieve this goal. We have observed that there are limitations to the range of its applicability when we consider the level of precision we initially pursuit. However, when the model fits well, results concerning its predictive abilities are encouraging.

Experimenting with variations in the setup for obtaining a deeper understanding of situations where our proposal and USL can be applied are needed. Another issue concerns the variability of virtual-machine performance in clouds [7]. Further investigations on how the effect of virtual machines on predictability are needed. Such investigations and experiments are planned for the near future.

Acknowledgement

This research is supported by CAPES - process BEX-1110-/14-4.

References

1. Aljohani, A., Holton, D., Awan, I.: Modeling and Performance Analysis of Scalable Web Servers Deployed on the Cloud. 2013 Eighth International Conference on Broadband and Wireless Computing, Communication and Applications pp. 238–242 (Oct 2013)
2. Bacigalupo, D., Jarvis, S., He, L., GR, N.: An investigation into the application of different performance prediction techniques to e-commerce applications. In: Proc. of the 18 th International Parallel and Distributed Processing Symposium (IPDPS). vol. 00 (2004)
3. Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: a survey. IEEE Transactions on Software Engineering 30(5), 295–310 (May 2004)
4. Chapman, C., Emmerich, W., Marquez, F.: Elastic service management in computational clouds. CloudMan 2010 pp. 1–8 (2010)
5. Chen, Y., Sun, X.h.: STAS: A Scalability Testing and Analysis System. In: IEEE International Conference on Cluster Computing. pp. 1–10 (2006)
6. Courtois, M., Woodside, M.: Using regression splines for software performance analysis. In: Proceedings of the second international workshop on Software and performance - WOSP '00. pp. 105–114 (2000)
7. Dejun, J., Pierre, G., Chi, C.: Resource Provisioning of Web Applications in Heterogeneous Clouds. In: USENIX Conference on Web Application Development (2011)
8. Dejun, J., Pierre, G., Chi, C.H.: Autonomous resource provisioning for multi-service web applications. In: Proceedings of the 19th international conference on World wide web - WWW '10. New York, New York, USA (2010)
9. Gao, J., Pattabhiraman, P., Bai, X., Tsai, W.T.: SaaS performance and scalability evaluation in clouds (Dec 2011)
10. Gong, Z., Gu, X., Wilkes, J.: PRESS: PRedictive Elastic reSource Scaling for cloud systems. In: Proceedings of the 2010 International Conference on Network and Service Management, CNSM 2010. pp. 9–16 (2010)
11. Grama, A.Y., Gupta, A., Kumar, V.: Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures. IEEE Parallel & Distributed Technology 1(3), 12–21 (1993)
12. Guerra, E., Moura, P., Besson, F., Rebouas, A., Kon, F.: Patterns for testing distributed system interaction. In: Conference on Pattern Languages of Programs (PLoP) (2014)
13. Gunther, N.: A Simple Capacity Model of Massively Parallel Transaction Systems. In: CMG-CONFERENCE (1993)

14. Gunther, N.: Guerrilla Capacity Planning. Springer, Berlin Heidelberg (2007)
15. Gunther, N.: A General Theory of Computational Scalability Based on Rational Functions. arXiv preprint arXiv:0808.1431 pp. 1–14 (2008)
16. Happe, J., Westermann, D., Sachs, K., Kapová, L.: Statistical Inference of Software Performance Models for Parametric Performance Completions. In: Research into PracticeReality and Gaps, pp. 20—-35. No. 216556, Springer (2010)
17. Harbaoui, A., Dillenseger, B., Vincent, J.M.: Performance characterization of black boxes with self-controlled load injection for simulation-based sizing. In: French Conference on Operating Systems (CFSE) (2008)
18. Harbaoui, A., Salmi, N., Dillenseger, B., Vincent, J.M.: Introducing Queuing Network-Based Performance Awareness in Autonomic Systems. Sixth International Conference on Autonomic and Autonomous Systems pp. 7–12 (Mar 2010)
19. Jogalekar, P., Woodside, M.: Evaluating the Scalability of Distributed Systems. In: Thirty-First Hawaii International Conference on System Sciences. vol. 7, pp. 524 – 531 (1998)
20. Jogalekar, P., Woodside, M.: Evaluating the scalability of distributed systems. IEEE Transactions on Parallel and Distributed Systems 11(6), 589–603 (2000)
21. Klems, M., Bermbach, D., Weinert, R.: A runtime quality measurement framework for cloud database service systems. In: Proceedings of the 8th International Conference on the Quality of Information and Communications Technology (2012)
22. Lee, J.Y., Lee, J.W., Cheun, D.W., Kim, S.D.: A Quality Model for Evaluating Software-as-a-Service in Cloud Computing (2009)
23. Lim, H., Babu, S., Chase, J., Parekh, S.: Automated control in cloud computing: challenges and opportunities. In: Proceedings of the 1st workshop on Automated control for datacenters and clouds. pp. 13–18 (2009)
24. Menascé, D.A.: Capacity Planning : An Essential Tool for CAPACITY. IEEE IT Professional (August), 33–38 (2002)
25. Moura, P., Kon, F.: Automated scalability testing of software as a service. In: 8th International Workshop on Automation of Software Test (AST). pp. 8–14 (May 2013)
26. Salmi, N., Dillenseger, B., Harbaoui, A., Vincent, J.m., Labs, O.: Model-based Performance Anticipation in Multi-tier Autonomic Systems : Methodology and Experiments. International Journal on Advances in Networks and Services 3(3), 346–360 (2010)
27. Snellman, N., Ashraf, A., Porres, I.: Towards Automatic Performance and Scalability Testing of Rich Internet Applications in the Cloud. In: 37th EUROMICRO Conference on Software Engineering and Advanced Applications. pp. 161–169 (Aug 2011)
28. Srinivas, A., Janakiram, D.: A model for characterizing the scalability of distributed systems. ACM sigops operating systems review (2005)
29. Sun, X.H., Chen, Y.: Reevaluating Amdahl's law in the multicore era. Journal of Parallel and Distributed Computing 70(2), 183–188 (2010)
30. Tchana, A., Dillenseger, B., Palma, N.D., Etchevers, X., Vincent, J.M., Salmi, N., Harbaoui, A.: Self-scalable Benchmarking as a Service with Automatic Saturation Detection. In: Middleware 2013. pp. 389–404 (2013)
31. Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., Tantawi, A.: An analytical model for multi-tier internet services and its applications. ACM SIGMETRICS Performance Evaluation Review 33(1), 291 (Jun 2005)
32. Vasić, N., Novaković, D., Miucin, S., Kostić, D., Bianchini, R.: DejaVu: Accelerating Resource Allocation in Virtualized Environments. In: Seventeenth International

Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2012)